**PROIECT DISCIPLINA POO**

[ Joc de șah în consolă ]

Student: Catargiu Ștefan-Iliuță

CUPRINS

Cuprins

TEMA PROIECT

# TEMA ȘI MOTIVAȚIA ALEGERII

**Tema**

Joc simplu de șah în consolă.

Un joc simplu de tip Chess Game în consolă consolidat din mutarea pe rând a pieselor până când unul dintre jucători pierde.

**Motivatia alegerii**

Am ales aceasta tema deoarece mi-a oferit posibilitatea excelenta de a intelege conceptele fundamentale ale dezvoltarii de jocuri. Am lucrat cu elemente precum gestionarea starii unei piese, mutarea ei și înțelegerea regulilor jocului de șah.

Dezvoltarea unui joc de șah în consolă poate fi o sarcină destul de grea, mai ales pentru începători. Această temă oferă oportunitatea de a se confrunta cu aspecte tehnice complexe, cum ar fi implementarea regulilor de joc, gestionarea mișcărilor și reprezentarea tablei de joc. Această temă va aduce o provocare tehnică interesantă și va spori competențele de programare în C++.

Există mai multe motive pentru care ai putea alege tema "joc simplu RPG în consolă" pentru proiectul tău de programare. Iată câteva motive comune:

1. Utilizarea conceptelor OOP: Implementarea unui joc de șah în C++ permite aplicarea eficientă a principiilor programării orientate pe obiecte (OOP). Crearea claselor pentru piese, tablă de joc și jucători facilitează organizarea modulară a codului și permite extinderea și întreținerea simplă a proiectului.
2. Dezvoltarea abilităților de gândire logică: Implementarea unui joc de șah necesită abilități de gândire logică și planificare strategică. Proiectul va încuraja dezvoltarea algoritmilor de evaluare a poziției și a strategiilor de joc, îmbunătățind astfel abilitățile de gândire analitică și logică, care sunt esențiale în dezvoltarea software.

Un joc simplu de șah în consolă poate fi ușor de rulat pe diferite platforme și poate fi accesibil pentru utilizatori cu orice nivel de expertiză în jocuri de gândire. Consola oferă un mediu de rulare consistent și nu necesită instalarea de software suplimentar sau cerințe hardware specifice, permițând astfel un public mai larg să se bucure de jocul tău.

CAPITOLUL I

# ELEMENTE TEORETICE

# DESCRIEREA PROBLEMEI

Scopul jocului: Jocul are ca scop învingerea celuilalt jucător ori prin șah mat ori când ambii jucători rămân fără mutări disponibile și jocul se termină în remiză.

Sistem de mutare a pieselor: Sistemul de mutare a pieselor se folosește de coordonatele de bază a piese și noile coordonate date de către jucător pentru .

Interacțiunea cu utilizatorul: Interacțiunea cu utilizatorul în jocul de șah poate fi realizată prin intermediul unei interfețe grafice. Utilizatorul va putea alege și efectua mutările pieselor folosind comenzi specifice jocului de șah.

Gestionarea starii jocului: Jocul de șah nu necesită un sistem de salvare și încărcare a progresului deoarece este un joc de strategie de durată scurtă și se joacă într-o singură sesiune. Jucătorii pot finaliza jocul într-o singură ședință și nu este necesară păstrarea progresului între sesiuni.. De asemenea va exista o gestionare a erorilor pentru a asigura o experienta de joc fara probleme si pentru a oferi un feedback adecvat utilizatorului in cazul introducerii unor comenzi incorecte.

# ABORDAREA TEORETICA A PROBLEMEI

* + 1. Structura generala a jocului
       - Definirea claselor si entitatilor principale: Jucatorul, Piesele și Tabla De Joc.
* Definirea Jucatorului:

Definirea jucatorului se face printr-un bool. Atunci când acel bool returnează TRUE, jucătorul este culoarea alb, iar atunci când returnează FALSE, jucătorul este culoarea negru. Un exemplu:

tabla.adaugaPiesa(new Tura(0, 0, true, &tabla));

tabla.adaugaPiesa(new Tura(0, 7, false, &tabla));

* + - * + Definirea claselor:

Definirea claselor se face printr-o clasa de bază denumita Piesa. Aceasta se va folosi de o funcție denumită isMoveValid si de get și set pentru a facilita mutările pieselor. Tot aici avem și funcția de afișare pe matricea bidimensională.

class Piesa

{

protected:

int x, y;

bool alb;

public:

Piesa(int x, int y, bool alb)

{

this->x = x;

this->y = y;

this->alb = alb;

}

int getX()

{

return x;

}

int getY()

{

return y;

}

bool isAlb()

{

return alb;

}

void setX(int x)

{

this->x = x;

}

void setY(int y)

{

this->y = y;

}

virtual bool isMoveValid(int x, int y) = 0;

virtual ~Piesa() {}

virtual void afisare() = 0;

};

* + - * + Definirea clasei TablaDeJoc

Această clasă gestionează tabla de joc și conține funcții pentru afișarea tablei, adăugarea pieselor, mutarea pieselor și verificarea stării jocului.

class TablaDeJoc

{

private:

Piesa\* piesa[8][8];

public:

TablaDeJoc()

{

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

piesa[i][j] = nullptr;

}

}

}

void afisareTabla()

{

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

if (piesa[i][j] != nullptr)

{

piesa[i][j]->afisare();

}

else

{

cout << "\_ ";

}

}

cout << endl;

}

}

void adaugaPiesa(Piesa\* piesa)

{

this->piesa[piesa->getX()][piesa->getY()] = piesa;

}

Piesa\* getPiesa(int x, int y)

{

return piesa[x][y];

}

void mutarePiesa(int x1, int y1, int x2, int y2)

{

std::swap(piesa[x1][y1], piesa[x2][y2]);

}

bool isInSah(bool alb)

{

// Determina pozitia regelui jucatorului cu culoarea specificata

int xRege = -1, yRege = -1;

for (int x = 0; x < 8; ++x)

{

for (int y = 0; y < 8; ++y)

{

Piesa\* piesa = getPiesa(x, y);

if (piesa != nullptr && piesa->isAlb() == alb)

{

xRege = x;

yRege = y;

break;

}

}

}

// Verifica daca exista vreo piesa a adversarului care poate muta pe pozitia regelui

for (int x = 0; x < 8; ++x)

{

for (int y = 0; y < 8; ++y)

{

Piesa\* piesa = getPiesa(x, y);

if (piesa != nullptr && piesa->isAlb() != alb && piesa->isMoveValid(xRege, yRege))

{

return true;

}

}

}

return false;

}

bool areMutariValide(bool alb)

{

// Verifica daca exista cel putin o piesa a jucatorului cu culoarea specificata care are cel putin o mutare valida

for (int x = 0; x < 8; ++x)

{

for (int y = 0; y < 8; ++y)

{

Piesa\* piesa = getPiesa(x, y);

if (piesa != nullptr && piesa->isAlb() == alb && piesa->isMoveValid(x,y))

{

return true;

}

}

}

return false;

}

bool isInSahMat(bool alb)

{

// se verifica daca regele este in sah

if (!isInSah(alb))

{

return false;

}

// se verifica daca se poate face o mutare pentru a iesi din sah

for (int x = 0; x < 8; ++x)

{

for (int y = 0; y < 8; ++y)

{

Piesa\* piesa = getPiesa(x, y);

// se verifica daca piesa exista si apartine jucatorului cu culoarea specificata

if (piesa != nullptr && piesa->isAlb() == alb)

{

// se parcurg toate mutarile valide ale piesei

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

if (piesa->isMoveValid(i, j))

{

// se incearca mutarea piesei pe noua pozitie

mutarePiesa(x, y, i, j);

// se verifica daca regele nu mai este in sah

bool esteInSah = isInSah(alb);

// se revine la starea initiala a tablei

mutarePiesa(i, j, x, y);

if (!esteInSah)

{

return false;

}

}

}

}

}

}

}

// daca nu s-a putut face nicio mutare pentru a iesi din sah, este sah mat

return true;

}

bool existaPiesa(int x, int y)

{

Piesa\* piesa = getPiesa(x, y);

return (piesa != nullptr);

}

};

* + - * + Definirea fiecare piese în parte:

Definirea fiecărei piese în parte se face prin clasele numite Pion, Regina, Rege, Nebun, Cal și Tura. Fiecare cu regulile sale specifice pentru mutare.

class Pion :public Piesa

{

bool primaMutare;

public:

Pion(int x, int y, bool alb, TablaDeJoc\* tabla) :

Piesa(x, y, alb),

tabla(tabla),

primaMutare(false)

{

}

bool isMoveValid(int x, int y) override

{

// Verifică dacă există alte piese în pozițiile (x+1, y-1) și (x-1, y-1)

if (x == getX() + 1 && (y == getY() - 1 || y == getY() + 1))

{

if (tabla->existaPiesa(x, y) && tabla->getPiesa(x, y)->isAlb() != isAlb())

{

// Verifică dacă mutarea este pe verticală

if (getX() == x)

{

// Verifică dacă este o mutare înainte pe verticală pentru piesele albe

if (isAlb() && y == getY() - 1 && !tabla->existaPiesa(x, y))

{

return true;

}

// Verifică dacă este o mutare înainte pe verticală pentru piesele negre

if (!isAlb() && y == getY() + 1 && !tabla->existaPiesa(x, y))

{

return true;

}

}

// Verifică dacă este o mutare pe diagonală pentru a captura o piesă de culoare opusă

if (x == getX() + 1 && (y == getY() - 1 || y == getY() + 1))

{

if (tabla->existaPiesa(x, y) && tabla->getPiesa(x, y)->isAlb() != isAlb())

{

return true;

}

}

// Nu s-a îndeplinit nicio condiție de mutare validă, returnează false

return false;

}

}

if(primaMutare)

{

primaMutare = false;

if(getX() == x)

{

if(getY() + 1 == y) return true;

if(getY() + 2 == y) return true;

return false;

}

return false;

}

else return getX() == x && getY() + 1 == y;

}

void afisare() override

{

if (alb)

{

std::cout << "P";

}

else

{

std::cout << "p";

}

}

private:

TablaDeJoc\* tabla;

};

* + - * + Clasa Regina

class Regina : public Piesa

{

public:

Regina(int x, int y, bool alb, TablaDeJoc\* tabla) : Piesa(x, y, alb), tabla(tabla) {}

bool isMoveValid(int x, int y) override

{

int dx = x - getX();

int dy = y - getY();

// daca mutarea e in diagonala

if (std::abs(dx) == std::abs(dy))

{

int stepX = dx > 0 ? 1 : -1;

int stepY = dy > 0 ? 1 : -1;

int currX = getX() + stepX;

int currY = getY() + stepY;

while (currX != x && currY != y)

{

if (tabla->getPiesa(currX, currY) != nullptr)

{

return false;

}

currX += stepX;

currY += stepY;

}

return true;

}

// daca mutarea e in sus/jos sau stanga/dreapta

if (dx == 0 || dy == 0)

{

int step = dx != 0 ? dx / std::abs(dx) : dy / std::abs(dy);

int currX = getX() + step;

int currY = getY() + step;

while (currX != x || currY != y)

{

if (tabla->getPiesa(currX, currY) != nullptr)

{

return false;

}

currX += step;

currY += step;

}

return true;

}

return false;

}

void afisare() override

{

if (alb)

{

std::cout << "Q";

}

else

{

std::cout << "q";

}

}

private:

TablaDeJoc\* tabla;

};

* + - * Clasa Rege:

class Rege : public Piesa

{

public:

Rege(int x, int y, bool alb, TablaDeJoc\* tabla) : Piesa(x, y, alb), tabla(tabla) {}

bool isMoveValid(int x, int y) override

{

int dx = std::abs(x - getX());

int dy = std::abs(y - getY());

// regele poate fi mutat cu o casuta in orice directie

if (dx <= 1 && dy <= 1)

{

// se verifica daca destinatia este goala sau ocupata de o piesa a adversarului

Piesa\* piesa = tabla->getPiesa(x, y);

return piesa == nullptr || piesa->isAlb() != alb;

}

return false;

}

void afisare() override

{

if (alb)

{

std::cout << "R";

}

else

{

std::cout << "r";

}

}

private:

TablaDeJoc\* tabla;

};

* + - * Clasa Nebun:

class Nebun :public Piesa

{

public:

Nebun(int x, int y, bool alb, TablaDeJoc\* tabla) : Piesa(x, y, alb), tabla(tabla) {}

bool isMoveValid(int x, int y) override

{

if (std::abs(x - getX()) == std::abs(y - getY()))

{

int dirX = (x > getX()) ? 1 : -1;

int dirY = (y > getY()) ? 1 : -1;

for (int i = getX() + dirX, j = getY() + dirY;

i != x || j != y;

i += dirX, j += dirY)

{

if (tabla->getPiesa(i, j) != nullptr)

{

return false;

}

}

return true;

}

return false;

}

void afisare() override

{

if (alb)

{

std::cout << "N";

}

else

{

std::cout << "n";

}

}

private:

TablaDeJoc\* tabla;

};

* + - * Clasa Cal:

class Cal : public Piesa

{

public:

Cal(int x, int y, bool alb, TablaDeJoc\* tabla) : Piesa(x, y, alb), tabla(tabla) {}

bool isMoveValid(int x, int y) override

{

int dx = std::abs(x - getX());

int dy = std::abs(y - getY());

return (dx == 2 && dy == 1) || (dx == 1 && dy == 2);

}

void afisare() override

{

if (alb)

{

std::cout << "C";

}

else

{

std::cout << "c";

}

}

private:

TablaDeJoc\* tabla;

};

* + - * Clasa Tura:

class Tura : public Piesa

{

public:

Tura(int x, int y, bool alb, TablaDeJoc\* tabla) : Piesa(x, y, alb), tabla(tabla) {}

bool isMoveValid(int x, int y) override

{

if (x == getX() || y == getY())

{

int dirX = (x > getX()) ? 1 : -1;

int dirY = (y > getY()) ? 1 : -1;

for (int i = getX() + dirX, j = getY() + dirY;

i != x || j != y;

i += dirX, j += dirY)

{

if (tabla->getPiesa(i, j) != nullptr)

{

return false;

}

}

return true;

}

return false;

}

void afisare() override

{

if (alb)

{

std::cout << "T";

}

else

{

std::cout << "t";

}

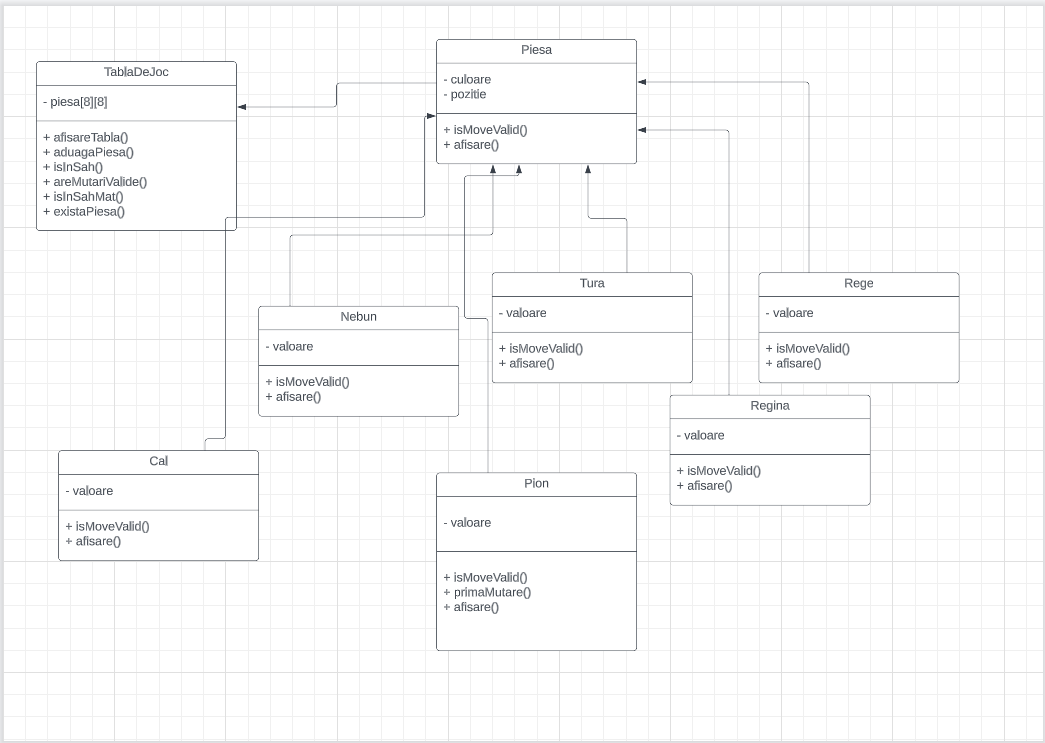
}

private:

TablaDeJoc\* tabla;

};

* + - * Stabilirea relatiilor si interactiunilor dintre entitati



# ELEMENTE SPECIFICE POO

* + 1. Clase și obiecte:

- Definirea claselor pentru entitățile principale ale jocului, cum ar fi piesele si tabla de joc.

- Crearea obiectelor specifice fiecărei clase și gestionarea interacțiunilor dintre ele.

1.3.2. Encapsulare:

- Utilizarea encapsulării pentru a ascunde detaliile și implementarea internă a claselor și a funcționalității acestora.

- Folosirea accesului controlat la metode și variabile pentru a proteja și a gestiona interacțiunea cu obiectele.

1.3.3. Moștenire:

- Utilizarea moștenirii pentru a crea clase derivate din clase de bază, cum ar fi crearea unor clase pentru piesele de joc fiecare cu mutările sale specifice conform regulilor jocului de șah.

- Implementarea unei ierarhii de clase pentru a permite reutilizarea și extinderea funcționalității.

1.3.4. Polimorfism:

- Utilizarea polimorfismului pentru a permite utilizarea unor metode cu același nume, dar cu implementări specifice pentru fiecare clasă.

- Definirea metodelor virtuale și folosirea lor în clasele derivate pentru a suprascrie comportamentul din clasele de bază.

1.3.5. Asociere și agregare:

- Gestionarea relațiilor între obiecte prin intermediul asocierii și agregării, de exemplu, clasa TablaDeJoc are o asociere cu fiecare clasă specifică pieselor de șah

1.3.7. Polimorfism parametric (generice):

- Utilizarea generice pentru a crea clase și metode care pot fi parametrizate cu diferite tipuri de date, de exemplu, crearea unei clase generice pentru gestionarea unei liste de obiecte.

CAPITOLUL II

# IMPLEMENTARE

# TEHNOLOGII FOLOSITE

Tehnologiile utilizate în construcția proiectului includ limbajul de programare C++ și mediul de dezvoltare integrat (IDE) Code::Blocks. C++ este un limbaj de programare puternic și versatil, cunoscut pentru performanța sa și abilitatea de a dezvolta aplicații de înaltă performanță. Alegerea limbajului C++ pentru acest proiect este potrivită pentru dezvoltarea unui joc simplu RPG, oferind control detaliat asupra resurselor și o flexibilitate în implementarea logicii jocului.

Pentru a gestiona dezvoltarea și codificarea, am utilizat mediul de dezvoltare integrat Code::Blocks. Code::Blocks este un mediu IDE popular pentru dezvoltarea aplicațiilor C++, oferind o interfață intuitivă și un set de instrumente pentru editare de cod, depanare și compilare. Acest mediu IDE oferă funcționalități esențiale pentru dezvoltarea proiectului, cum ar fi evidențierea sintaxei, completarea automată a codului și un sistem de construire și depanare eficient.

Utilizarea limbajului C++ împreună cu mediul de dezvoltare Code::Blocks oferă o bază solidă și eficientă pentru dezvoltarea jocului tău RPG în consolă, permițând implementarea mecanicilor de joc, gestionarea datelor și interacțiunea cu utilizatorul într-un mod eficient și ușor de înțeles.

# DIAGRAMA DE CLASE, SCHEMA BLOC, WORKFLOW

